134 CodeLab

January 18, 2017

1 Getting Started

1.1 Installing Python (Anaconda)

We recommend using the Anaconda Python distribution, as it bundles together many useful packages (including numpy, matplotlib, and IPython) that we will use throughout this class. To install Anaconda, please see here. Make sure you get the version compatible with your OS (Mac, Linux, or Windows). If you'd like a more comprehensive document, please check out CS109's guide here.

1.2 Using iPython Notebooks

In order to follow along, we ask that you open this file (`Codelab_0.ipynb`, also available on the site) in IPython and make changes to your local copy. To do this, please first download the file, navigate to the file's directory in Terminal, and then call `ipython notebook [filename]`. To execute code in each cell, press `Shift + Enter`.

These codelabs are in IPython because it is easy to interact with and provides nice compartmentalization when teaching various concepts, but the programming assignments will require you to write your own Python scripts. These are files that end with the extension `.py`, e.g., `test.py`. You can use any text editor to create these files; popular ones include Sublime Text, Notepad++ (Windows only), Atom, Emacs, and Vim.

In order to run the script `test.py`, open Terminal and navigate to its directory. From the directory, type `python test.py`. All the basic syntax and structure of the code remains the same; it's just a different way of packaging it.

1.3 Python Shell

The Python Interactive Shell is a light-weight way to explore Python. It's commonly used for debugging, quick tasks, and as a calculator! To get to the shell, open Terminal and type `python`. More instructions are located here.

2 Python Basics

2.1 Lists

Lists are basic Python structures that function as arrays. They can store all types of data. You can index a specific element using `[]` brackets, which comes in handy very frequently.

In [ ]:
```
# Variable assignment
a = 3
b = 2
print a + b  # 5

c = []
d = [1,2]
```

In [ ]:
```
# List basics
print 'List basics:'
list_a = []
list_b = [1,2,3,4,5,6]
print 'list b:', list_b
second_elt_in_list_b = list_b[1]  # remember, we index from 0
print 'second element in list b:', second_elt_in_list_b
tmp = [1,3,4]
sub_list_list_b = [list_b[i] for i in tmp]
print 'sub list:', sub_list_list_b
list_b.append(7)
print 'new list b:', list_b
last_elt_b = list_b[-1]
last_elt_b_2 = list_b[len(list_b)-1]
print 'last element two ways:', last_elt_b, last_elt_b_2
print '\n'  # newline character

# List of lists
print 'List of lists:'
list_of_lists_c = [list_a,list_b]
print 'list of lists c:', list_of_lists_c
list_of_lists_c.append([3,4,5])
print 'new list of lists c:', list_of_lists_c
third_list_in_c = list_of_lists_c[2]
print 'third list in c:', third_list_in_c
second_elt_of_third_list_in_c = list_of_lists_c[2][1]
print 'second element of third list in c:', second_elt_of_third_list_in_c
print '\n'
```

In [ ]:
```
# Given a list of lists, do the following:
LL = [[1,2],[2,3],[3,4,5],[4,5,6,7],[5,6],[6,7,8,8,9]]

# a. Get the third sublist

# b. Get the first element of the first sublist

# c. Get the second element of the third sublist

# d. Append the list [7,8] to LL

# e. Append the value 8 to the fourth sublist in LL
```

2.2 Conditional Statements and Loops

Conditional statements and loops are useful control structures that allow you to structure the way your program will execute. Note that Python cares about whitespace; proper indentation after conditional statements and function definitions (in the next section) is key to avoiding syntax errors, even in the Interactive Shell.

```
In [ ]: # Conditional statements
        if 1 > 0:
            print '1 > 0'

        if 1 < 0:
            print '1 < 0'
        else:
            print '1 !< 0'

        if 1 < 0:
            print '1 < 0'
        elif 1 < 2:
            print '1 < 2'
        else:
            print 'none of the above'
```

```
In [ ]: # For loops
        for i in range(0,10):
            print i

        for i in xrange(0,10):
            print i

        # looping through lists
        for x in list_b:
            print x

        # While loops
        j = 0
        while j < 10:
            print j
            j += 1
```

```
In [ ]: # Using LL from the previous example, do the following:

        # a. Create a new list first_LL of the first elements of each sublist

        # b. Create a new list last_LL of the last elements of each sublist

        # c. Print all values in all sublists in the range [4,8]

        # d. Print all sublists of length 3 or greater
```

2.3 Functions

Functions are useful ways of encapsulating blocks of code to perform tasks. Each function is declared with the `def` keyword and takes a list of arguments. Additionally, most functions return something via the `return` keyword, which exits the function.

```
In [ ]: # Functions are declared with the 'def' keyword followed by the
        # function name and a list of arguments to the function.
        def my_function(arg1,arg2):
            tmp = arg1 + arg2
            return tmp

        return_value = my_function(1,3)
        print 'sum:', return_value

        def sum_list_naive(L):
            sum = 0
            for x in L:
                sum += x
            return sum

        print 'sum list:', sum_list_naive([1,4,5,3])
```

2.4 Dictionaries

In Python, dictionaries are essentially hash tables. They store (key,value) pairs and can contain data of all types.

```
In [ ]: # Dictionary example
        my_dict = {}  # declare empty dictionary
        my_dict['key1'] = 'value1'  # insert ('key1','value1') pair in dictionary
        print 'my dict:', my_dict  # prints entire dictionary
        print 'first entry:', my_dict['key1']  # index into dictionaries with keys

        # Second dictionary example
        dict1 = {}
        for x in xrange(1,21):
            # each dictionary entry is a list of all whole numbers less than that v
            dict1[x] = range(0,x)
        print 'list dict:', dict1
        print 'last entry:', dict1[20]
```

2.5 File I/O

In order to load a file (`f`) from a script (`s`), you need the relative positions of the two files. For example, if both are in the same folder (call it `parent`), then you can simply call `open(f)` inside `s`. However, if the files are not in the same folder, then you may have to change the relative (or absolute) path you use to distinguish where the files are in relation to each other. For example, if the `parent` directory containing a script has a `data` folder with a file (`f`), then in order to open

`f` from `s`, you have to call `open(./data/f)` in order to explicitly tell Python that the data to open is contained within a folder in the current directory. A single dot (`./`) represents the current folder, and two dots (`../`) represents the folder one level up (i.e., the folder containing the current folder).

In the `Codelab_0/data` folder, there is a dataset called `iris.txt`. This section will show you how to read the data into Python.

```
In [ ]: # Read file naively
        with open('iris.txt', 'rb') as iris:
            next(iris)  # skip header line
            iris_data = []  # declare list of lists for data
            for entry in iris:
                # Strip '\n' characters and split entries on commas
                entry_data = entry.rstrip().split(',')
                # Convert numeric entries to floats instead of strings
                entry_data[0] = float(entry_data[0])
                entry_data[1] = float(entry_data[1])
                entry_data[2] = float(entry_data[2])
                entry_data[3] = float(entry_data[3])

                iris_data.append(entry_data)

            print iris_data
```

```
In [ ]: # Use pandas
        import pandas as pd

        # read data into a Pandas dataframe
        iris_df = pd.read_csv('iris.txt', sep=",", header=False)
        print iris_df
```

In case you want to learn more about the Pandas framework, which is great for dealing with large datasets, please click here. Its dataframes have interesting syntax associated with them, and CS 109 has a nice guide to Pandas here. Other links this guide points to are here and here.

2.6 Tricks and Syntactic Sugar

```
In [ ]: # List comprehensions
        list_b = [1,3,4,2,5,6,4,5,7,5,6,4,5,3,1,2]

        # Normal for loop
        tmp1 = []
        for x in list_b:
            if x > 3:
                tmp1.append(x)
        print 'Normal for loop:', tmp1

        # List comprehension
```

```
        tmp2 = [x for x in list_b if x > 3]
        print 'List comprehension:', tmp2
```

```
In [ ]: # Checking membership

        # Lists
        list_d = [1,2,3,4,5]

        if 1 in list_d:
            print '1 is in list d'

        if 6 not in list_d:
            print '6 is not in list d'

        # Dictionaries
        # dict.keys() and dict.values() are lists of the keys and values in the dic
        dict_a = {'a': 3, 'b': 2, 'c': 1}

        if 'a' in dict_a.keys():
            print 'a is a key in dict a'
        if 'd' not in dict_a.keys():
            print 'd is not a key in dict a'

        if 3 in dict_a.values():
            print '3 is a value in dict a'
        if 4 not in dict_a.values():
            print '4 is not a value in dict a'
```

2.7 Exercises

Write a function that goes through the following list and finds the smallest element. Do not use any built-in functions such as min(L).

```
In [ ]: # Find the smallest element of this list
        L = [3,1,2,4,5,3,1,4,2,5,6,3,6,7,5,4,7,1,2,4,3,5,6,7,5]

        def find_smallest(input_list):
            # TODO
            return 0  # change this
```

Given the following list of lists (LL), create a new list of the minimum elements in each list. For this function, please use the `find_smallest` function you defined above in the first exercise, not the built-in `min` function. If your find_smallest is buggy, however, feel free to use `min`.

```
In [ ]: # List of lists
        LL = [[0,1,3,4,5,2,4,3,1],[3,5,4,6,3,2,4,7,6],[3,3,3],[1,3,2,6],...
              [1,3,0,2,5,7],[4,2,7,6,8],[6,3,5,4,8],[1,2,1,5,0],[9,9,6,4,7]]

        def create_smallest_list(list_of_lists):
```

```
        # TODO
        return 0  # change this
```

Now, find the overall smallest element in a list of lists. You should be able to do this with `find_smallest` and `create_smallest_list`!

```
In [ ]: # List of lists
        LL = [[0,1,3,4,5,2,4,3,1],[3,5,4,6,3,2,4,7,6],[3,3,3],[1,3,2,6],...
              [1,3,0,2,5,7],[4,2,7,6,8],[6,3,5,4,8],[1,2,1,5,0],[9,9,6,4,7]]

        def find_global_smallest(LL):
            # TODO
            return 0  # change this
```

3 Useful Python Libraries

3.1 Random

We will frequently need to simulate random variables in constructing random graphs, conducting simulations, etc. Python has a number of libraries for generating randomness. The most basic is the random library. For simulating draws from a random variable, Numpy (see below) is more commonly used.

```
In [2]: import random

        random.seed(42) # seeding the RNG 'fixes' the randomness so that runs after
                        # useful for debugging random code

        i = random.randint(1, 10) # randomly pick an integer in [1,10]
        print 'Randomly selected %d' % i

        sample = random.sample(range(10), 3) # randomly pick three integers without
        print 'Randomly selected ' + ', '.join([str(j) for j in sample])

        f = random.random() # return a float in [0, 1)
        print 'Randomly selected %.3f' % f

        u = random.uniform(1,10) # randomly pick a float from [1, 10]
```

```
Randomly selected 7
Randomly selected 0, 2, 1
Randomly selected 0.736
```

3.2 Numpy

NumPy is a popular numerical Python library used for scientific computing. It is included in the standard Anaconda installation package. You can read more about it here. Numpy is particularly useful for dealing with large datasets, which we won't do too often in this class, but Numpy is

still a good package to be aware of. Here is a good tutorial, while in this lab we highlight some of Numpy's random functions.

```
In [ ]: import numpy as np

        ints = np.random.random_integers(1,10, (2,2))
        print 'Randomly sampled these ints: '
        print ints

        normals = np.normal(0, 1, (2,2))
        print 'Randomly sampled values from iid normals: '
        print normals

        binoms = np.binomial(5, .5)
        print 'Randomly sampled this value from a binomial distribution: %.3f' % bi

        expos = np.exponential(1)
        print 'Randomly sampled this value from an exponential distribution: %.3f'
```

3.3 Plotting

```
In [ ]: %matplotlib inline

        import matplotlib
        from matplotlib import pyplot as plt  # use matplotlib's pyplot package
        import seaborn as sns  # use seaborn to make it pretty

        plt.plot([1,3,4],[3,6,7])  # plt.plot draws a line
        plt.show()
```

```
In [ ]: # Get data to plot
        sepal_length_data = [e[0] for e in iris_data]
        sepal_width_data = [e[1] for e in iris_data]

        # Plotting sepal width vs. sepal length from iris
        plt.figure(1)  # declare a figure
        ax1 = plt.subplot(111)
        plt.scatter(sepal_length_data, sepal_width_data)  # plt(x,y) plots x vs. y
        plt.xlabel('sepal length (cm)')  # label x-axis
        plt.ylabel('sepal width (cm)')  # label y-axis
        plt.title('Sepal Width vs. Sepal Length')
        plt.show()
```

```
In [ ]: # Plot histogram of sepal width data
        plt.figure(1)
        ax2 = plt.subplot(111)
        plt.hist(sepal_width_data, bins=15)
        plt.title('Histogram of Sepal Widths')
        plt.xlabel('sepal width')
```

```
        plt.ylabel('frequency')
        plt.show()
```

3.4 Exercises

You are given two lists of data corresponding to x- and y- values to plot. They are organized such that the n^{th} value in each lists correspond to each other. Plot all corresponding (x,y) values where the x-value is strictly greater than 5.

```
In [ ]: # How to generate random lists, just as an aside; we will use hard-coded va
        x_vals_rand = [random.randint(0,10) for v in xrange(0,20)]
        y_vals_rand = [random.randint(0,10) for w in xrange(0,20)]

        # X and Y lists
        x_vals = [5, 10, 1, 1, 9, 0, 6, 7, 4, 4, 6, 7, 5, 9, 4, 2, 6, 10, 8, 2]
        y_vals = [5, 8, 3, 6, 7, 5, 6, 3, 2, 10, 10, 0, 7, 7, 8, 1, 6, 5, 0, 10]
```

4 Graph Theory

4.1 Representing Graphs

The most common data structures for representing graphs are adjacency matrices and adjacency lists. In general, adjacency matrices are better for dense graphs (because they take up $O(n^2)$ room), and they support very quick edge lookups. However, it's slow to iterate over all edges. Adjacency lists are better for sparse graphs and allows quick iteration over all edges in a graph, but finding the existence of a specific edge takes longer. For these examples, we will focus on using adjacency lists.

The most basic way of representing a graph is through a list of lists. In this way, we can say that the n^{th} list represents all of the n^{th} vertex's connections. We will go through an example on the board.

```
In [ ]: # List of lists representation of a graph
        # Graph G: Vertices = {0,1,2,3}, Edges = {(0,1),(1,2),(2,3),(3,0)}

        G_LL = [[1,3],[0,2],[1,3],[0,2]]

        # Vertex 0's neighbors
        print G_LL[0]

        # Vertex 1's neighbors
        print G_LL[1]
```

Another way of representing a graph is through a dictionary. See the below section for a quick introduction to a dictionary. The keys in this case will be the vertex names, and the corresponding values will be a list of all adjacent vertices. Note that using a dictionary is more convenient for graphs that have strings as labels for each vertex, but using a dictionary is more expensive in terms of memory and time. For the example above, we have the following.

```
In [ ]: # Dictionary representation of a graph
        # Graph G: Vertices = {0,1,2,3}, Edges = {(0,1),(1,2),(2,3),(3,0)}

        G_dict = {0:[1,3], 1:[0,2], 2:[1,3], 3:[0,2]}

        # Vertex 0's neighbors
        print G_dict[0]

        # Vertex 1's neighbors
        print G_dict[1]
```

4.2 Traversing Graphs with BFS

Breadth first search (BFS) is a common and useful graph traversal algorithm. A graph traversal algorithm describes a process for visiting the nodes in a tree or graph, where traversing from one node to another is only allowed if there is an edge connecting those vertices. A motivational property of BFS is that it can find connectivity and shortest distances between two nodes in graphs. For this class, we will apply BFS to compute centrality measures and other graph properties. In this lab we will study how BFS runs, how to implement it, and how it can be used.

4.2.1 Queues

Before we can begin to study BFS, we need to understand the queue data structure. Similar to the non-computer science understanding of the word, a queue is essentially a line. Data, in our case nodes or vertices, enter the queue and are stored there. The operation of adding data to a queue is called enqueue() or push(). The order in which they enter the queue is preserved. When we need to retrieve data, we take the earliest element from the queue and it is removed from the queue. This operation is called dequeue() or pop(). This is referred to as first-in-first-out (FIFO).

In Python, we can implement a queue naively as a list. However, lists were not built for this purpose, and do not have efficient push() and pop() operations. Alternatives include implementations included in libraries (which we will be using) or defining your own class and operations. For this lab we will use the queue data structure from the collections library.

```
In [ ]: from collections import deque

        queue = deque(["Scarlet", "White", "Mustard"])
        queue.append("Plum")
        queue.popleft()
        queue.append("Greene")
        queue.popleft()
        queue.popleft()
        murderer = queue.popleft()

        # who is Boddy's murderer?
```

4.2.2 Algorithm

Recall that a graph traversal is a route of nodes to follow or traverse, where we can only move between two nodes if there is an edge between them. Breadth first search specifies a process for

doing so. In the simplest implementation, BFS visits a given node, adds its neighbors to the queue, then visits the element returned by the queue. To begin the BFS, we start at a given node, then visit all of that node's nieghbors, then visit all of one neighbor's neighbors, then another neighbor's neighbors, etc. Whenever we visit a node for the first time, we mark it as visited; otherwise we do nothing when re-visiting a node. The algorithm terminates once we have visited everything reachable from the given node, and we have a list of nodes reachable from that node.

```
In [ ]: # graph G represented as an adjacency list dictionary, a node v to start at
        def simpleBFS(G, v):
            # TODO: implement BFS!
```

```
In [ ]: # G as an adjacency list
        G = {1:[2,3], 2:[5], 3:[4], 4:[], 5:[2,3], 6:[5]}
        print simpleBFS(G, 1)
```